





Relacionando as camadas Repository, Controller e Entity no Spring Boot

O Spring Boot se caracteriza por organizar o código em diferentes camadas. Ao longo desse módulo, veremos as camadas Repository, Controller e alguns relacionamentos mais complexos, aplicando-os na camada Entity. Além disso, veremos como essas três camadas se relacionam no framework em questão.

No Spring, a camada Repository é responsável pela persistência dos dados, trafegados ao longo da aplicação através da(s) instância(s) de uma ou mais Entidades no banco de dados. Logo, todas as operações relacionadas ao banco ficam sob responsabilidade dessa camada – sejam elas de recuperação, persistência, atualização ou deleção de registros. Diferente das Entities, no Repositório temos interfaces ao invés de classes. Além disso, cada Entidade deverá possuir sua própria interface na camada em questão, ou seja, se criarmos uma classe chamada Aluno na camada Entity, deveremos então criar uma interface, na camada Repository, chamada AlunoRepository. Repare, aqui, na convenção de se utilizar a palavra “Repository” como sufixo ao nome da Entidade. Isso ajuda a identificar, pelo nome do arquivo, a qual entidade um repositório se refere.

Outra característica importante e bastante conveniente do Spring Boot é permitir que nosso repositório implemente interfaces de repositório pré-existentes, como a JpaRepository, a CrudRepository e a PagingAndSortingRepository. Cada uma dessas interfaces



possui uma série de métodos pré-implementados, dos quais poderemos fazer uso ao declarar nossa interface e informar que estenderemos uma delas. Em geral, utilizamos a JpaRepository (que, por sua vez, já estende às outras duas). Com isso, temos disponíveis vários métodos de manipulação de dados, como os relacionados ao CRUD (findById, findAll, save, delete, etc.), além de recursos como paginação e outros.

O código abaixo demonstra a implementação do repositório relativo à Entidade Aluno – à esquerda temos o código da Entity e à direita o do Repository.

```
@Entity
@Table(name="aluno")
public class Aluno{
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name = "aluno_id")
   private Integer alunoId;

   @Column(name = "aluno_nome")
   private String alunoNome;

   @ManyToOne
   @JoinColumn(name="turma_id")
   private Turma turma;

   //get´ss e set´s
}
```

```
import
org.springframework.data.jpa.repository.JpaRepository;

public interface AlunoRepository extends
JpaRepository<Aluno, Integer>{
}
```

Como podemos perceber, há uma relação interdependente entre a camada Entity e a Repository: ambas são necessárias e trabalham em conjunto para que os dados (a serem persistidos ou recuperados do banco de dados) transitem em nossa API. Na hierarquia de camadas do Spring Boot, podemos dizer que essas duas camadas são as de mais "baixo nível", ou as que ficam mais próximas ao banco de dados. Por outro lado, temos então a

camada que, seguindo essa mesma lógica, podemos chamar de camada de mais alto nível, a Controller.

A camada Controller, no Spring, contém classes responsáveis por responder a requisições HTTP/HTTPS. Logo, nela inserimos os recursos que queremos expor em nossa API/ e que queremos disponibilizar para serem utilizados por clientes, como outros sistemas de software, ou mesmo por outras aplicações em nosso próprio ecossistema, por exemplo, podemos ter um sistema composto por uma aplicação web e por um aplicativo mobile, em que ambos consomem serviços disponíveis em nossa API desenvolvida utilizando Spring, fazendo requisições sobre o protocolo HTTP, tendo como alvo a camada Controller. Isso posto, e seguindo o padrão do framework, devemos ter, não obrigatoriamente, mas de forma recomendada, uma classe na camada Controller para cada uma das entidades que compõem nossa API. Além disso, tais classes recebem algumas anotações especiais – anotações essas que, a nível de classe, informam que a classe em questão é um Controller do tipo REST (@RestController); e, a nível de recursos e serviços disponibilizados, representados pelos métodos que compõem a classe, o tipo de acesso a cada um deles (@GetMapping, @PostMapping, @PutMapping, @DeleteMapping e @PathMapping). Considerando o repositório que criamos anteriormente, vejamos agora o controller correspondente:

```
@RestController
@RequestMapping("/aluno")
public class AlunoController {
  @GetMapping
  public ResponseEntity<List<Aluno>> getAllAlunos() {
    //a implementar: chamada ao service correspondente
    return new ResponseEntity<>(listaAlunos, HttpStatus.OK);
  }

  @GetMapping("/{id}")
  public ResponseEntity<Aluno> getAlunoById(@PathVariable Integer id) {
    //a implementar: chamada ao service correspondente
    return new ResponseEntity<>(aluno, HttpStatus.OK);
  }

  @PostMapping
  public ResponseEntity<Aluno> saveAluno(@RequestBody Aluno aluno) {
    //a implementar: chamada ao service correspondente
    return new ResponseEntity<>(novoAluno, HttpStatus.CREATED);
  }

  @PutMapping("/{id}")
  public ResponseEntity<Aluno> updateAluno(@PathVariable Integer id, @RequestBody Aluno aluno) {
    //a implementar: chamada ao service correspondente
    return new ResponseEntity<>(alunoAtualizado, HttpStatus.OK);
  }

  @DeleteMapping("/{id}")
  public ResponseEntity<String> deleteAluno(@PathVariable Integer id) {
    //a implementar: chamada ao service correspondente
    return new ResponseEntity<>("", HttpStatus.OK);
  }
}
```

Como podemos ver ao analisar o código acima, o controller possui algumas características específicas, e já citadas anteriormente, a saber:

- A classe recebe duas anotações. Uma para informar que se trata de um Controller (a @RestController) e outra, opcional, para definir uma URI específica para o recurso (a @RequestMapping). Além disso, essa segunda anotação recebe como parâmetro o nome da URI – no exemplo temos: /aluno . Essa URI, combinada com a URL da API e mais o verbo HTTP referente ao endpoint que queremos acessar representam o endpoint para o qual as requisições deverão ser feitas.

- Nesse exemplo, foram disponibilizados serviços para cada operação CRUD. Perceba que para cada um deles foi designado, através da respectiva anotação, o verbo HTTP para acesso ao mesmo. Além disso, repare que alguns recebem um parâmetro adicional (/{id}). Nesse ponto, cabe destacar que não é possível ter dois serviços no Controller com uma mesma anotação de verbo HTTP. Repare nos dois primeiros métodos: o primeiro foi anotado apenas com @GetMapping, enquanto o segundo recebeu essa anotação e também o parâmetro, do tipo "path", /{id} . Se não tivéssemos declarado esse parâmetro, mantendo ambos os métodos apenas com o @GetMapping, seria retornado um erro ao rodarmos nossa API;

- Por fim, perceba que os códigos de cada método estão incompletos, já que não foram implementadas as chamadas para a camada Service.

Antes de finalizarmos esse módulo, voltaremos ao assunto Relacionamentos entre Entidades. Além dos relacionamentos mais simples, como one-to-one, one-to-many e many-to-one, é possível também representarmos relacionamentos mais complexos, como o many-to-many. A partir de agora, veremos a implementação, mais básica, desse tipo de relacionamento no Spring. Para começar, temos o seguinte DER:

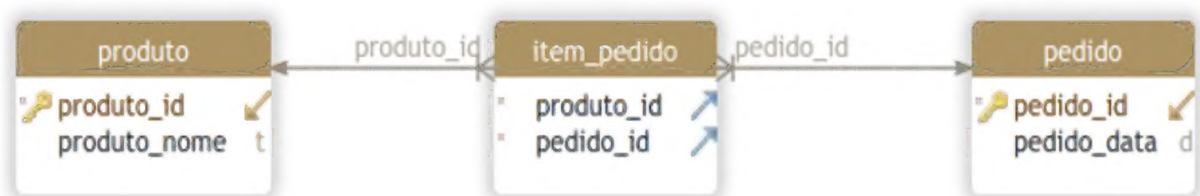


Figura 1: Relacionamento many-to-many no DER



A relação apresentada acima é a relação clássica, já bastante conhecida, de uma venda, em que temos o produto, o pedido de venda e a tabela de relação entre ambos – a item_pedido. Tal relação implica que um produto pode estar presente em diferentes pedidos e que um pedido pode ser composto por vários produtos. A tabela de ligação recebe apenas as chaves-primárias das tabelas relacionadas. Isso caracteriza tal relacionamento como muito-pra-muitos básico. Uma variação desse relacionamento, inclusive muito utilizada, consiste na inclusão de coluna ou colunas adicionais na tabela de relacionamento. Logo, cada variação dessas possui uma forma diferente de implementação no Spring. A seguir, veremos a forma mais simples:

```
@Entity
@Table(name = "produto")
public class Produto {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "produto_id")
        private Integer produtoId;

        @Column(name = "produto_nome")
        private String produtoNome;

        @ManyToMany
        @JoinTable(name = "item_pedido",
                joinColumns = @JoinColumn(name = "produto_id"),
                inverseJoinColumns = @JoinColumn(name = "pedido_id"))
        Set<Pedido> pedidos;

        //get´s e set´s
}
```



```
@Entity
@Table(name = "pedido")
public class Pedido {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "pedido_id")
        private Integer pedidoId;

        @Column(name = "pedido_data")
        private LocalTime pedidoData;

        @ManyToMany
        Set<Produto> produtos;

        //get´s e set´s
}
```

A princípio, a estrutura das entidades acima é bastante similar à estrutura de outras entidades em que temos outros tipos de relacionamento como OneToOne ou OneToMany. Entretanto, além da anotação @ManyToMany, há um detalhe extra a ser percebido nesse caso: a inclusão, em um dos lados do relacionamento (em uma das duas entidades que participam da relação) da anotação @JoinTable. Tal anotação recebe alguns parâmetros, cujos significados são descritos a seguir:

- name: nome da tabela de ligação, no banco de dados. Em nosso exemplo: item_pedido ;

- joinColumns: nome da coluna que representa a chave-estrangeira, na tabela de ligação, referente à entidade na qual a anotação está sendo inserida. Em nosso exemplo, como inserimos a anotação na Entidade Produto, o valor do parâmetro é “produto_id”;

- inverseJoinColumns: nome da coluna que representa a chave-estrangeira no outro lado da relação, ou seja, na tabela pedido. Em nosso caso, o valor do parâmetro é “pedido_id”.

Por fim, repare que não é necessário incluir a anotação @JoinTable nas duas entidades que participam do relacionamento. Inse[illegible]m apenas uma já permite ao Spring entender o relacionamento, assim como nos permite recuperar, a partir da Entidade Produto, todos os pedidos nos quais uma instância da mesma foi inserida e vice-versa.

Além das situações e casos mais comuns vistos até aqui, o Spring possui ainda alguns recursos avançados, fornecendo entre outras funcionalidades, diferentes formas de mapearmos a relação entre duas entidades. Nesse sentido, vejamos como funcionam as anotações @Embedded e @Embeddable. Para isso, vamos a um exemplo prático: imagine que, num determinado negócio, queremos armazenar os dados de uma empresa e de um contato da mesma, como segue:



Empresa
Id
Nome
Telefone
Nome do Contato
Telefone do Contato

Nesse caso, utilizando os recursos vistos até aqui, poderíamos separar a entidade empresa em duas outras entidades: Empresa e Contato, criando a relação entre ambas, com as respectivas anotações. Entretanto, há outra forma de separar essas entidades: a utilização das anotações @Embedded e @Embeddable, cujas definições seriam, como os nomes sugerem, incorporar uma classe em outra. Veja como ficariam os códigos dessas duas classes:



```
@Embeddable
public class Contato {

    private String nome;

    private String telefone;

    // get´s e set´s
}
```

```
@Entity
public class Empresa {

    @Id
    @GeneratedValue
    private Integer id;

    private String nome;

    private String telefone;

    @Embedded
    private Contato contato;

    // get´s e set´s
}
```

Repare nos códigos, que a classe Contato foi anotada com @Embeddable, informando se tratar de uma classe "incorporável". Já na classe Empresa, ao definir a instância de Contato, ela foi marcada como @Embedded, informando se tratar de uma instância incorporada.

Ainda sobre o recurso de incorporação, é importante tratar de outra funcionalidade do Spring: é possível criar a estrutura do banco de dados a partir da definição das entidades (seus atributos, relacionamentos e respectivas anotações). Em outras palavras, podemos tanto criar nossa API fazendo o mapeamento objeto-relacional de um banco de dados previamente existente, como criarmos e mantermos a estrutura do banco de dados a partir da própria API, mais precisamente da definição das entidades. Nesse

caso, criarmos a estrutura do banco a partir das entidades, a utilização das classes acima, Empresa e Contato, gerará um problema de conflito, visto que há dois atributos com o mesmo nome: “telefone”, um na classe Empresa e outro na classe Contato. Para solucionar tal questão, podemos acrescentar uma outra anotação, informando ao Spring que ele deverá sobrescrever o nome dos atributos ao criar as colunas. Veja abaixo como ficaria, então, a classe Empresa com essas novas anotações:

```
@Entity
public class Empresa {

    @Id
    @GeneratedValue
    private Integer id;

    private String nome;

    private String telefone;

    @Embedded
          @AttributeOverrides({
            @AttributeOverride( name = "nome", column = @Column(name = "contato_nome")),
            @AttributeOverride( name = "telefone", column = @Column(name = "contato_telefone"))
          })
    private Contato contato;

    // get´s e set´s
}
```

Através da anotação @AttributeOverrides conseguimos sobrescrever o nome do atributo, informando que um novo nome deverá ser utilizado na criação da coluna.

Tendo abordado o relacionamento ManyToMany e as anotações @Embeddable e @Embedded, chegamos ao final desse módulo. Além de tal conteúdo, tratamos de forma conceitual e teórica, sobre as camadas Repository e Controller.



Atividade Extra

Para ver outros exemplos de aplicação do relacionamento ManyToMany, sugiro que vejam o tutorial abaixo.

Link: https://www.baeldung.com/jpa-many-to-many **(Acesso em 14/09/2022)**

Referência Bibliográfica

WALLS, Craig. **Spring Boot in Action**. Shelter Island, NY: Manning, 2016.

Atividade Prática Módulo 3

Título da Prática: Criando os primeiros Repository e Controller.

Objetivos: Codificar uma interface Repository e uma Classe Controller.

Materiais, Métodos e Ferramentas: IDE (Spring Tool Suite; Jetbrains Intellij; etc)

Atividade Prática

A codificação de uma API Restful, utilizando o Spring Boot, segue um passo-a-passo: devemos criar, independente de por onde começarmos, as camadas (packages) e códigos – Entidades, Repositórios, Serviços e Controles que formarão a nossa aplicação.

Nesse contexto, chegou a hora de criar nossa primeira Interface Repositório e nossa primeira Classe Controller. Para tal, considere as seguintes informações:

- Entidades contidas na API: Instrutor e Turma

- Tipo de dado dos atributos @Id de ambas as entidades: Integer

- No Controller deverão ser criados dois métodos para cada classe:

  procurarPodId: receberá um inteiro como parâmetro e devolverá a instância da entidade correspondente ao id fornecido;

  salvar: receberá uma nova instância da entidade a ser armazenada no banco de dados;

- Os códigos das entidades estão disponíveis a seguir:



```
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonBackReference;
import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;

@Entity
@Table(name = "turma")
public class Turma {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id_turma")
private Integer idTurma;

@Column(name = "horario")
private Date horarioTurma;

@Column(name = "duracao")
private Integer duracaoTurma;
```

```
import java.util.Date;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;
import javax.validation.constraints.NotEmpty;

import com.fasterxml.jackson.annotation.JsonIdentityInfo;
import com.fasterxml.jackson.annotation.ObjectIdGenerators;

@Entity
@Table(name = "instrutor")
public class Instrutor {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "id_instrutor")
private Integer idInstrutor;

@Column(name = "rg")
private Integer rgInstrutor;

@Column(name = "nome")
private String nomeInstrutor;

@Column(name = "nascimento")
private Date dataNascimento;

@Column(name = "titulacao")
private Integer titulacaoInstrutor;

@OneToMany(mappedBy = "instrutor")
private List<Turma> turmaList;
```

Gabarito Atividade Prática

Os códigos disponibilizados a seguir refletem o que foi solicitado nessa atividade:

- Repository

```
import org.springframework.data.jpa.repository.JpaRepository;

//import *.entity.Turma;

public interface TurmaRepository extends JpaRepository<Turma,Integer> {
}
```



```
import org.springframework.data.jpa.repository.JpaRepository;
//import *.entity.Instrutor;

public interface InstrutorRepository extends
JpaRepository<Instrutor,Integer> {
}
```

- Controller



```
//imports ...

@RestController
@RequestMapping("/turma")
public class TurmaController {

@GetMapping("/{id}")
public Turma procurarPorId(@PathVariable Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Turma
}

@PostMapping
public Turma salvar(@RequestBody Turma turma) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Turma;
}
}
```

```
//imports ...

@RestController
@RequestMapping("/instrutor")
public class instrutorController {

@GetMapping("/{id}")
public Instrutor procurarPorId(@PathVariable Integer id) {
      //aqui deverão ser implementados, posteriormente, os códigos para
localizar por id
      //return Instrutor
}

@PostMapping
public Instrutor salvar(@RequestBody Instrutor instrutor) {
      //aqui deverão ser implementados, posteriormente, os códigos para
salvar a Entidade
      //return Instrutor;
}
}
```

Ir para exercício